LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-662983

A Scalable Prescriptive Parallel Debugging Model

N. B. Jensen, S. Karlsson, N. Q. Nielsen, G. L. Lee, D. H. Ahn, M. Legendre, M. Schulz

October 21, 2014

IEEE International Parallel & Distributed Processing Symposium
Hyderabad, India
May 25, 2015 through May 29, 2015

Disclaimer

A Scalable Prescriptive Parallel Debugging Model

Nicklas Bo Jensen[†], Niklas Quarfot Nielsen[†‡§], Gregory L. Lee[§],
Sven Karlsson[†], Matthew Legendre[§], Martin Schulz[§], Dong H. Ahn[§]

[†]*Technical University of Denmark, DTU Compute*
{*nboa, svea*}*@dtu.dk*

[‡]*Mesosphere Inc*
niklas@mesosphere.io

[§]*Lawrence Livermore National Laboratory, Computation Directorate*
{*lee218, legendre1, schulzm, ahn1*}*@llnl.gov*

***Abstract*—Debugging is a critical step in the development of any parallel program. However, the traditional interactive debugging model, where users manually step through code and inspect their application, does not scale well even for current supercomputers due its centralized nature. While lightweight debugging models, which have been proposed as an alternative, scale well, they can currently only debug a subset of bug classes. We therefore propose a new model, which we call prescriptive debugging, to fill this gap between these two approaches. This user-guided model allows programmers to express and test their debugging intuition in a way that helps to reduce the error space. Based on this debugging model we introduce a prototype implementation embodying this model, the DySectAPI, allowing programmers to construct probe trees for automatic, event-driven debugging at scale. In this paper we introduce the concepts behind DySectAPI and, using both experimental results and analytical modeling, we show that the DySectAPI implementation can run with a low overhead on current systems. We achieve a logarithmic scaling of the prototype and show predictions that even for a large system the overhead of the prescriptive debugging model will be small.**

I. Motivation

Debugging is an important capability for large-scale simulations, but little has changed in how we debug applications. At the same time, high-fidelity simulations continue to drive strong demand for extremely large-scale machines while pushing application complexity to extremes. As a net result of this trend, machines with over a million cores are not uncommon today [1], [2]; and further, mission-critical applications often comprise a few million lines of code, coupling many scientific packages and libraries written in a wide range of programming paradigms and languages, e.g., C, C++, FORTRAN and Python. This sheer scale combined with application complexity has made debugging one of the most arduous tasks in code-development for high performance computing, HPC.

This situation will become even more challenging in the future [3]. Due to power and energy concerns, performance gains on HPC systems no longer come from increased single-thread performance, but rather, from increased core counts and the use of accelerators or co-processors. This trend increasingly requires programmers to rely on hybrid programming models, such as MPI + OpenMP or MPI + CUDA, to realize the full hardware potential, but this comes at significantly added coding complexity and to many unintended side effects between the various models. In short, programming complexity will rise, and with that so will the likelihood that bugs, particularly with respect to parallelism, will be introduced into codes; the current interactive (per thread/process) debugging techniques are not sufficient in helping programmers overcome these challenges. Without effective and scalable debugging models, and the tools that embody these models, the cost of debugging will sharply increase due to the lost productivity of programmers and wasted compute cycles.

Unfortunately, while programming models designed to improve programmers' productivity are actively studied and proposed, there has been a lack of studies to understand the most capable debugging models with the same goal.

Traditional parallel debugging [4], [5], [6] is a several-decade-old model and does not scale even to today's core counts. Most importantly, manually stepping through source lines and inspecting the application state overwhelms the programmers with too much information. As a response to this problem, the lightweight debugging model [7], [8], [9], [10] arose, where information is sacrificed for scalability, but has the opposite problem – it scales well by design, but for many classes of errors, does not provide enough information. Further work has focused on fully automatic or semi-automatic debugging that helps automate the detection of suspicious behavior, e.g., through relative debugging [11] or automated anomaly detection [12], but these approaches also limit themselves to particular classes of bugs and do not provide the generality of interactive debuggers.

To fill this gap, we propose a novel scalable debugging model called prescriptive parallel debugging that helps users

automate the tedious parts of interactive debugging, while keeping its generality, and present our implementation that embodies this model: the *Dynamic Scalable Event-Chain Tracing API* or *DySectAPI*. This model offers a highly-scalable debugging paradigm by using an *engine* that *expresses a programmer's debugging intuition* to *automatically and progressively reduces the error search space* across both the task and source-code dimensions.

Our approach allows programmers to install debugging probes into a parallel application. These probes can gather data under user-specified conditions, in the form of either debugging procedures (e.g., trigger an action when a certain set of breakpoints is hit) or code behaviors (e.g., trigger an action when the code hangs). Probes are linked into a probe tree, automatically driving the prescribed debugging actions to reduce the error space. When the specified conditions are met, it presents highly condensed debug information to the programmer.

We demonstrate the scalability of our model by empirically evaluating the performance overheads of various probe-tree topologies of the DySectAPI and also by deriving a performance model. The evaluation suggests that using a probe tree that prunes out processes that are not of interest scales better than a flat topology analogous to the traditional debugging model. Further, we present results from a case study that show the effectiveness of our model: we used a DySectAPI probe tree to effectively debug a previously undiagnosed, real-world bug in a scientific application, which manifested itself only when scaled to 3,456 MPI processes.

A. Contributions

To summarize, we make the following contributions:

- A novel prescription-based debugging model striking a balance between scalability and capability;
- The DySectAPI, an implementation embodying this model;
- Empirical experiments and an analytic model of overheads of the DySectAPI;
- A case study on a real world application showing its effectiveness.

II. Models in Parallel Debugging

The current state of the art in parallel debugging is focused on four main models: traditional; lightweight; semi-automatic; and automatic. In this section, we discuss the pros and cons of these models and the implications of the architectural and application trends in HPC.

The traditional debugging model seeks to aid programmers in interactively diagnosing an error. It enables them to view the detailed program state and to modify it at any point in execution. The tools that offer this paradigm include GDB [6], DDT [5] and TotalView [4]. They must provide features that support nearly *all* debugging facets. Commonly, these features include various data displays, lock stepping of execution, breakpoints or evaluation-points, and process or thread group control, such as barriers, if the tool is parallel-aware.

This model is highly effective in isolating the root cause of many classes of errors at low to moderate scales. However, in the face of large concurrency and high application complexity, its effectiveness starts to decrease sharply. More importantly, manually viewing and managing detailed state information at various points of execution is becoming increasingly unwieldy at only a few thousands of processes and on complex modern design patterns. The recent efforts of several parallel debugger vendors [5], [13] have improved software scalability by supporting the model through innovative communication and display mechanisms. But its fundamental scheme of having to enable *all idioms* for interactive use and the central point of control, i.e., the user, clearly limits its scalability.

The lightweight model has recently emerged in specific response to the challenges at scale. This model pursues trade-offs between debugging scalability and capabilities. Thus, the tools of this paradigm, such as the Stack Trace Analysis Tool, STAT [7], [8], [9], [14], IBM's Blue Gene Coreprocessor debugger [10] and Cray's Abnormal Termination Processing, ATP [15], drastically limit the types and amount of execution state information that are fetched, collected and displayed. They also limit the level of user interaction and provide only coarse mechanisms to select points in execution to analyze. This paradigm has proven to be extremely scalable, for example, STAT has successfully isolated certain classes of errors that only emerge over one million processes. However, due to its coarseness, it can leave programmers with no actionable information on some other classes of errors.

The semi-automatic model is based on user-guided automation of finding errors in large data arrays. Relative debugging [11] allows users to select corresponding data arrays and points in two slightly different versions of the same program, and at runtime automatically differentiates the arrays at those points. The idea is that the region of code where corresponding arrays of different versions diverge is likely to be where the error originated. Because the bulk of analysis is *computed*, this model significantly reduces user interaction and saves the user from having to examine individual data array elements manually. However, its main debugging mode that requires two different simultaneously running jobs can hamper debugging those errors that emerge only at large scales, due to resource requirements.

Further, recent work, such as assertion-based parallel debugging [16], has advanced this model to be used in more diverse scenarios, including comparing processes within a single job, and also to be more scalable. However, it targets data-centric errors, as opposed to being general purpose.

The automatic model has burgeoned in recent years with a promise to isolate general programming errors with

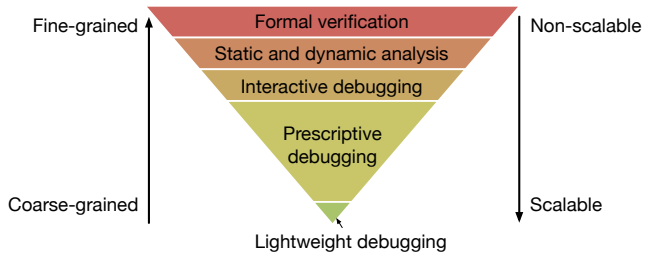


Figure 1: Debugging models.

no user intervention. Specific work in this area includes AutomaDeD [17], [18], [19], which applies statistical techniques on a semi-markov model representation of MPI processes to automatically detect and localize software bugs. The model also strives to automate code instrumentation and hence avoids opening the execution state and selection mechanisms to the users. While the automatic identification of erroneous tasks and code regions is the ideal approach, and recent work [18] has shown that this model can scale easily to a very large number of tasks, the level of isolation is only as good as the analysis and instrumentation techniques, which are currently still lacking. Often, a user's debugging intuition can allow a model to overcome the lack of analysis accuracy and precision, but the automatic nature makes it hard to take advantage of this intuition for error isolation. Other than these approaches, more popular form of automated tools tend to target specific types of errors, such as memory leaks [20] or MPI coding errors [21], [22], [23], [24].

III. In Search for Sweet Spots

At Lawrence Livermore National Laboratory (LLNL), one of the largest supercomputing centers, we have provided many tools embodying the debugging models described in Section II. In recent years, however, it became apparent that a significant gap exists in models of parallel debugging on large HPC environments. The traditional model offers a general-purpose environment, but does not scale well. Other debugging models scale better, but can target only specific classes of errors, making them special purpose.

For example STAT can be used to efficiently debug hangs at large scale, while doing so with a traditional debugger can be slow and complicated. STAT groups processes exhibiting similar behavior and can quickly expose why the hang arises, oftentimes due to a small set of outlier processes. With a traditional debugger, digesting the information from the many processes can overwhelm the programmer. On the other hand, a simple bug in the program logic that is exhibited even at small scale can be simpler to debug with a traditional debugger. STAT is a special purpose tool and can only find a bug if it visible by analyzing the stack traces of the application.

Therefore, we find significant needs for a new model that can be extremely scalable, yet capable enough, to serve general-purpose debugging. Figure 1 illustrates this notion.

To explore the trade-off space, we first identify several key principles that should drive the construction of a new scalable, general-purpose debugging model.

The new model must:

1) Be an engine to test the hypothesis behind a programmer's debugging intuition;
2) Enable users to express their intuition easily in terms of progressive reduction of error space;
3) Guide run-time debug actions with minimal interaction with the user;
4) Present condensed information after the error space is reduced.

The first principle states that any general-purpose debugging model must assist users with mechanisms well-suited to test the programmer's debugging intuition, otherwise, the model is limited to be special purpose. The second principle dictates how to capture this intuition. Ultimately, the goal of any parallel debugger is to narrow down the root cause of a problem in several dimensions. One is to determine which particular process(es) or which thread(s) exhibit erroneous behavior. A second is to pinpoint the precise source-code location of the bug. Yet another dimension can be to isolate any contaminated data that led to the error. The new model must enable users to encode the notion of progressive reduction on the potentially huge error search space along all dimensions.

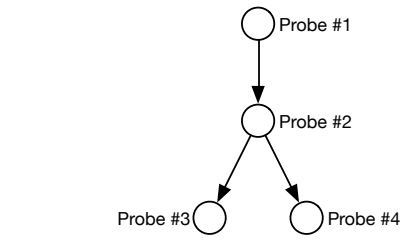
The third principle argues for the batch processing of debugging actions whereby the debugging expressions given by the programmer are evaluated in batches [25]. This is necessary because the sheer volume of debug data may slow down a traditional model to the point where the tool would be intolerable to use. Even if a debugger could process that data with interactive latencies, it would be challenging to present that much data in a form that is scalable to the user. Computers, on the other hand, are much better suited for the task of sifting through massive amounts of data. Along with the third principle, the fourth one addresses the limitation in a programmer's information processing ability. Any information provided to the users must not be too much for a human to digest.

IV. A New Model: Prescriptive Debugging

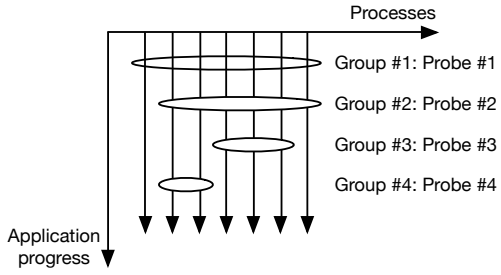
The traditional debugging paradigm has survived because it provides the rudimentary operations that a user needs to effectively reduce the error search space. In a typical debug session, a user first sets a breakpoint at a particular code location. Once that breakpoint is triggered, the user will evaluate the state of the application and subsequently set another breakpoint, perhaps on a subset of processes that satisfy certain conditions. This process is then repeated until the bug is isolated.

Our new prescriptive debugging model aims to capture the flexibility and generality of this interactive process, but allow users to codify individual steps and sequences in the form of debug probes that can then be executed without the need for individual interactions between debugger and user. Similar to Aspect-Oriented Programming, AOP [26], the prescriptive debugging model addresses the separation of concerns. AOP breaks down program logic into distinct concerns, where one concern could be debugging. AOP does not address scalability or debugging in general and does not satisfy the four guiding principles from section III. Essentially, the prescriptive debugging model provides the means for a user to codify their debugging intuition into prescribed debug sessions. The application can then be submitted into the system's batch queue to be run under that debug session. At runtime, the debugger follows the user's intuition by executing the debug probes and, at the end, scalably gathers summary information that can be examined by the user during the execution or at their convenience after the job has completed.

Our prescriptive parallel debugging model is built upon the notion of probes that can be linked together into a probe tree. A probe itself is composed of a domain, events, conditions, and actions as defined below.



(a) Probe tree.



(b) Process search space reduction using a probe tree.

Figure 2: Example probe tree and corresponding processes being filtered out during the search space reduction.

$$Probe = \langle Domain, Events, Conditions, \{Actions\}, \{Probes\} \rangle$$

The *domain* is the set of processes to install a probe into. It also includes a synchronization operation that determines how long the probe should wait for processes in the domain before proceeding. More precisely, after the first process triggers a probe, the remaining processes have until some specified timeout to participate.

We define an *event* as an occurrence of interest. Events borrowed from traditional debuggers include breakpoints, which specify a code location (when reached, the debugger will stop the target process) and data watchpoints, which monitor particular variables, memory locations or registers. An event can also be a user-defined timeout that instructs a probe to be triggered after some elapsed amount of time. Events can also capture asynchronous occurrences such as a program crash, a signal being raised or a system-level event such as memory exhaustion.

These events allow programmers to express their debugging in terms of a set of procedures and in terms of code behaviors—e.g., on detecting a hang or slowness.

Further, individual events can also be composed together to enable advanced fine-grained event selection.

When an event occurs, its associated *condition* is evaluated. The condition is an expression that can be evaluated either locally on each backend or globally across the domain. A local condition may, for instance, check if a variable equals a particular value. A global condition can evaluate an aggregated value, such as minimum, maximum or average, across the entire domain. Conditions can also be composed to specify multiple variables of interest or to combine local and global evaluations.

If the condition is satisfied, the probe is said to be triggered, and the specified *actions* are executed. Probe actions can be formulated by the user as an aggregation or a reduction, for example, aggregated messages, merged stack traces or the minimum and maximum of a variable.

A probe can optionally include a set of child probes, which is enabled upon the satisfaction of the parent probe's condition. In this manner, a user can create a probe tree. A probe tree naturally matches the control-flow traversal that is typical of an interactive session with a traditional debugger. This can effectively narrow down the search space across the source-code dimension.

An example of a probe tree is shown in Figure 2a and the corresponding search-space reduction is shown in Figure 2b. As the application progresses, the probe tree effectively narrows down the search across the process space. As child probes are only installed in processes that satisfy the condition, processes that are not of interest are implicitly filtered out. Filtering not only helps narrow down the debug search space, but also reduces the number of subsequent probe installations, the amount of tool communication and the volume of data produced. These qualities are paramount to the scalability of the prescriptive parallel debugging

model, both for the user's comprehension and for the tool's operation.

The process-space reduction has an additional benefit to debugging capabilities. While operations on the full application or a large set of processes should be lightweight in order to scale, operations on a small subset can be more complex. Thus, a well-formed probe tree would start with high-level summary information, such as a merged stack trace or a message displaying aggregated values, and get progressively complex, perhaps even gathering individual variable values across a subset of tasks.

A debugging session is then defined as a set of probe trees. Generic debug sessions can be created for common errors such as hangs, segmentation violations or other crashes. Furthermore, developers can create application-specific debugging sessions for their users to employ when an error occurs. With inside knowledge, the developer can write debug sessions that track known invariants for deviation, monitor control flow for abnormal behavior or specify conditions under which an application is considered to be hung. The aggregated debug log messages or merged stack traces can then be analyzed by the user or the developer to aid in identifying the root cause.

V. DySectAPI: The Dynamic Scalable Event Tracing API

Based on the guiding principles from Section III and following the prescriptive debugging model introduced above, we developed the Dynamic Scalable Event-Chain Tracing API, DySectAPI. DySectAPI allows programmers to express and test their debugging intuition, executes with minimal user interaction and only presents condensed information once the error space has been reduced.

The programmer specifies a debugging session prior to the execution, based on their debugging intuition. They do so in a session file using our API for expressing probes and probe trees.

The workflow of DySectAPI has six steps:

1) Prior to execution, the developer encodes a debugging session, which may contain several probe trees.
2) The session is compiled into a shared library.
3) The target application is launched by the DySectAPI runtime and the probe trees are installed in the specified domains.
4) The application executes.
5) When probes are triggered, the developer-specified conditions and actions are performed.
6) The condensed diagnostic output is presented to the programmer.

Figure 5 illustrates this workflow.

A. Expressing Debugging Intuition

DySectAPI allows programmers to express their debugging intuition using probes and probe trees. Probes are represented as C++ objects, which can be linked into a probe tree. A debugging session snippet can be seen in Figure 3. This snippet is compiled into a shared library and launched together with the target binary under the tool's control. The two probes are triggered at the source code location `instrumentationHead` and `instrumentPoint2` in the target example program in Figure 4. The first probe will be installed in all target processes. When triggered it will send a message through the network with the number of triggered processes. This will result in a message being printed by the frontend. In addition, when the first probe has been triggered, the second probe is enabled in the processes that triggered the first probe.

```
1  Probe* p1= new Probe(
2    Code::location("instrumentationHead"),
3    Domain::world(10),
4    Act::trace("probe 1: Location is
         instrumentationHead()"));
5
6  Probe* p2 = new Probe(
7    Code::location("instrumentPoint2"),
8    Domain::world(10),
9    Act::trace("probe 2: Location is
         instrumentationPoint2()"));
10
11 ProbeTree::addRoot(p1);
12 p1->link(p2);
```

Figure 3: Example probe tree debugging session.

```
1  void instrumentPoint2() {
2    static volatile int _count = 0;
3    _count++;
4  }
5
6  void instrumentHead() {
7    static volatile int _count = 0;
8    instrumentPoint2();
9    _count++;
10 }
11
12 int main() {}
13   MPI_Init(&argc, &argv);
14   for(int i = 0; i < N; i++) {
15     instrumentHead();
16     MPI_Barrier(MPI_COMM_WORLD);
17   }
18   MPI_Finalize();
19 }
```

Figure 4: Example parallel application.

The probe-tree debugging primitives allow programmers to specify their debugging intuition and to test their debugging hypothesis. The supported capabilities are:

Events: can be either a code-centric classical breakpoint or an asynchronous event such as a signal. The latter

includes a crash, an exit or a specific signal number.

Conditions: are evaluations based on data such as variables, e.g. `x > 0 && y < 0`.

Domain: specifies in which processes to install the probe. This can be all processes or in a subset based on their MPI rank. An optional timeout can also specify the maximum time to wait for other participants before aggregating. The default is to wait infinitely long or until all processes have been triggered.

Actions: can be formulated as aggregations or reductions, including messages that can aggregate the `min`, `max` and `desc`([min,max]) of variables, and the `function` or source-code `location` where triggered. There are no restrictions with respect to encapsulation and local variables. Further, the system can produce merged stack-traces both in text and graphical form.

B. Infrastructure

One important factor in DySectAPI is its scalability, which we achieve by basing communication on MRNet [27], an efficient tree-based overlay network, TBON, for scalable tool communication and data processing. This system allows us to not only execute communication hierarchically following a tree structure, but also to embed processing operations into the tree, avoiding a central processing bottleneck at the tool frontend. In particular, our aggregation process has two steps:

1) Local data is aggregated on each backend. A backend is attached to a set of processes. When one process triggers a probe, other processes triggered within a timeout will form one packet to be sent through the MRNet tree network.
2) Packages from multiple backends are aggregated. Each MRNet node is also setup to wait for a specified timeout before forwarding packets through the network.

This process is illustrated in Figure 6. A set of events happens on each backend and is then efficiently aggregated using the MRNet network. DySectAPI uses Dyninst [28] to control and debug application processes. Dyninst is an API for binary analysis, binary instrumentation and process control. Dyninst allows us to debug unmodified application processes.

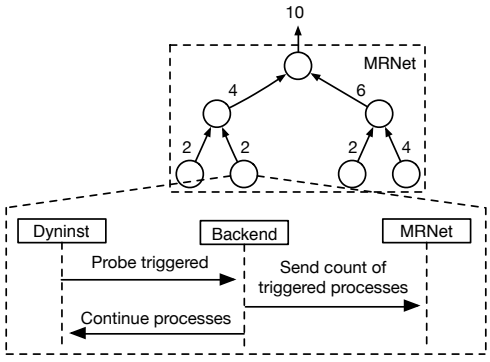


Figure 6: Communication architecture. The example probe counts the number of triggered processes and aggregates the information efficiently using the MRNet communication tree.

VI. Evaluation

We evaluate the DySectAPI implementation to demonstrate the scalability and effectiveness of the proposed prescriptive debugging model. However, evaluating all aspects of a debugger is hard. It is possible to measure the performance of the debugging primitives in a traditional interactive debugger, but those numbers would not account for human interaction, which would require a larger psychological study. Similarly, it is hard to quantify the usability of a debugger, given the complexity of debugging.

We therefore want to show that the prescriptive parallel debugging model has the scaling characteristics of the lightweight debugging model and a sufficient set of capabilities from the traditional, interactive debugger to capture a wide range of parallel bugs. The main questions we seek to answer are:

- What is the scalability of the prescriptive debugging model?
- Can we use the capabilities of the prescriptive debugging model to debug real parallel bugs only manifesting themselves at scale?

We answer the first question by focusing on how the underlying implementation scales on large parallel machines using a performance study combined with modeling that predicts the scalability beyond current machine resources. This is valuable to determine if our debugging model is going to scale on large systems. As a baseline for the scalability of the traditional interactive parallel debugger we ignore the human factor and use a batch debugging session that reflects the operations in a traditional interactive session.

Second, we wish to demonstrate the prescriptive debugging model validity and usefulness. We do so by focusing on the usability of the debugging model with a use-case example. In this way we show how the prescriptive debugging model can be applied to an undiagnosed real-world bug that emerges at large scale and outline the steps involved.

A. Experimental Setup

All experiments were conducted on the Cab Linux cluster located at the Lawrence Livermore National Laboratory. This cluster consists of 1,296 nodes, each with 2 Intel Xeon E5-2670 processors for a total of 20,736 cores and 41.5 TB memory. Each node has a total of 16 cores. DySectAPI has been built on top of MRNet 4.1 and Dyninst 8.2.

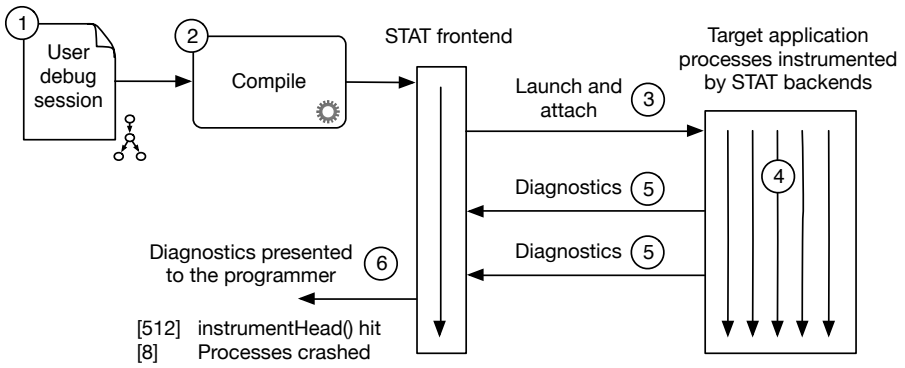


Figure 5: DySectAPI workflow: the programmer's debugging session is compiled and then launched. Probe trees are installed into the application and whenever probes are triggered, diagnostics are propagated up to the programmer.

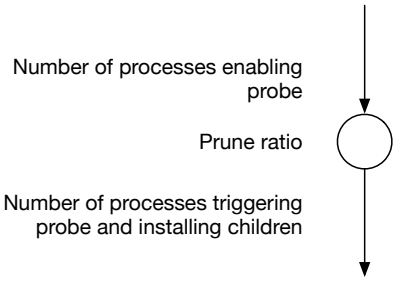


Figure 7: Pruning ratio in a probe.

B. Analytical Performance modeling

We derive an analytical performance model for an arbitrary probe tree to both reason about the underlying scalability and to predict scalability beyond current machine resources.

In each probe, we include a pruning factor, which is the fraction of processes that a probe filters out. A $ratio$ in a probe of 0.5 means that 50% of the processes are filtered out by that probe. Figure 7 illustrates this notion.

Each compute node runs several application processes and one backend daemon that is responsible for debugging all processes on that node. We refer to the ancestor of a probe, in the probe tree as $ancs(probe)$. The $max(installs_{backend}(probe))$ is the maximum number of installations on any backend for that probe.

We first consider the number of probe installations, $installs(probe)$, for a single probe on a single backend. The root probe is installed into all specified processes, while the number of children probes depends on how many processes satisfied the conditions associated with their ancestors:

$$installs(probe) = \begin{cases} \text{root}, max(installs_{backend}) \\ \text{otherwise}, invocs(ancs(probe)) \end{cases} \quad (1)$$

The number of invocations, $invocs(probe)$, defines the number of times a probe is triggered on a single backend:

$$invocs(probe) = (1 - ratio(probe)) \cdot installs(probe) \quad (2)$$

Probe installation are distributed across the backends and thus each backend install probes in parallel. For example if a probe has to be installed in a total of 16 processes across two backends, and they each install in 8 processes, the cost of doing so should only account for one backend, as the installations will happen simultaneously.

The cost of installation might be slightly higher on one backend due to variations in the load in each backend, therefore we use max to represent the maximum value that we encountered across a large number of runs.

The model is not restricted to how many installations happen on each backend, where one backend could perform all the installations. Therefore, we use the backend that has the highest number of installations. The cost of installation for one probe is:

$$\begin{aligned} cost_{inst}(probe) = & max(cost_{inst}) \cdot \\ & max(insts_{backend}(probe)) \end{aligned} \quad (3)$$

The cost of invoking the probe has a sequential part and a communication part. The latter assumes that the depth of the MRNet topology increases as the number of processes increases, making network cost a logarithmic function as it depends on the MRNet tree depth [29]. We define $N_{processes}$ as the number of processes that a probe has triggered in the application.

$$\begin{aligned} cost_{invoc}(probe) = & max(cost_{invoc}) \cdot \\ & max(invocs_{backend}(probe)) + \\ & max(cost_{network}) \cdot log(N_{processes}) \end{aligned} \quad (4)$$

The total cost of all probes in the tree is the sum of the costs for installing and invoking each probe.

$$cost_{total} = \sum_{i=0}^{|probes|} cost_{install}(i) + cost_{invoc}(i) \quad (5)$$

We are interested in deriving what limits the scalability of our model. Therefore, we study what happens when $\lim_{N_{processes} \to \infty}$. In this case the logarithmic network term dwarfs the other factors. The cost of installation, $cost_{inst}(probe)$ becomes a constant as both $max(cost_{inst})$ and $max(insts_{backend}(probe))$ are constants. Similarly, for the cost of invocation $cost_{invoc}(probe)$, both $max(cost_{invoc})$ and $max(invocs_{backend}(probe))$ are constants. This leaves the logarithmic network term as the dominating factor:

$$\lim_{N_{processes} \to \infty} cost_{total} = max(cost_{network}) \cdot log(N_{processes}) \quad (6)$$

Therefore we conclude that our model is able to achieve logarithmic scalability with our implementation. This is critical to enable scaling to extreme system sizes.

C. Performance Results

The model predicts logarithmic scalability $\mathcal{O}(\log n)$. In practice, many details can limit the scalability and we therefore seek to validate our model and to model scalability using experimental data.

Optimally we would compare the performance directly to a traditional debugger, however doing so would require a large psychological study. Therefore we ignore the human factor and use a flat probe tree without any pruning as the baseline. In the flat probe tree, the root probe installs four children probes without any pruning. Each child probe aggregates a single message across all processes that satisfy the probe's condition. An example of the probes can be seen in Figure 3, containing the first two probes in the tree, and the target application in Figure 4.

During execution of a DySectAPI debugging session, a reduction in the task search space naturally occurs as probes are dynamically enabled only when a specified condition is met, which leads to reductions in the amount of instrumentation and in the amount of debug information generated. A chained probe tree with each probe tree having a pruning ratio of 50% represents the prescriptive debugging model. The pruning ratio is chosen as a representative pruning ratio. Actual pruning ratios in real scenarios will depend on the prescribed debugging session, the program being debugged and the inputs to the program. We will later study how important the pruning ratio is. The pruning of processes will be spread out equally over all the backends in our experiment.

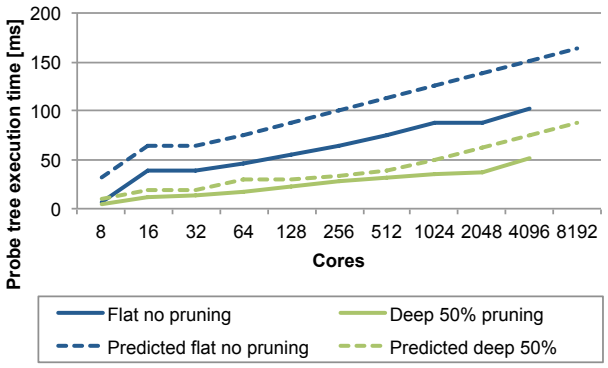


Figure 8: Actual and modeled execution time on Cab with 16 cores per node for a flat and a deep probe tree with 50% pruning.

Using microbenchmarking of DySectAPI we have obtained the following costs $cost_{invoc} = 0.72ms$, $cost_{install} = 0.28ms$ and $cost_{network} = 4.6ms$.

Figure 8 shows the actual and modeled overhead of the two probe trees with a pruning factor of 50% in the deep probe tree. We see that the modeled execution time for the deep probe tree more closely resembles the actual execution time, while for the flat tree there is a small difference. This is due to overlapping communication that results in a smaller communication overhead than modeled. In both cases, though, the model predicts an upper bound and therefore matches the observed scaling behavior, which is, as modeled, logarithmic.

The filtering of processes also reduces the amount of information presented to the programmer. For example in the chained deep tree consisting of four probe 87.5% of the original processes are filtered out.

D. Predicting Large Scale Performance

The introduced performance model is a good estimate of the worst-case runtime as exhibited by DySectAPI. We can use the model to predict the performance beyond current machine resources if we assume that the Cab cluster were an order of magnitude larger and exhibited the same performance.

Figure 9 show what the scalability would be like on a very large system. The modeled probes are the same ones as in Figure 8, consisting of four probes organized either in a flat tree or a deep chained tree with 50% pruning. The predicted overhead for a system with over 1,000,000 cores is just 180 ms for the four probes. This is consistent with the traditional interactive debugger DDT, which claims in the best case being able to step and display 700,000 cores in 100 ms [5].

We also model the performance of the DySectAPI implementation for multiple pruning ratios and for cases when the pruning is not spread out equally over the backends.

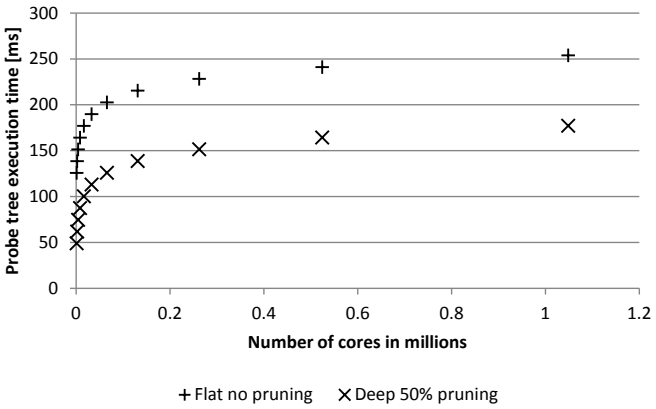


Figure 9: Modeled execution time on a larger version of Cab with 16 cores per node.

The pruning ratio affects the number of probe invocations per backend and has a demonstrable impact on overhead. At 1,048,576 cores the modeled overhead for the four probes organized in a deep chained tree, with 25% pruning in each of the probes, is 212.5 ms. With a pruning ratio of 50% the overhead is 177.2 ms and with a pruning ratio of 75% the overhead is only 135.7 ms. Thus, overhead can be reduced by expressing probe trees in a way that prunes out many processes.

Intuitively, and according to the performance model, the cost of installing and invoking a probe depends on the backend with the highest cost. If we assume that the pruning of processes is unbalanced such that none are pruned on one backend, at 1,048,576 cores for the four probes organized in a deep chained tree, with a 25% pruning ratio, the modeled execution time overhead is 240.6 ms. With a pruning ratio of 50% the overhead is 221.9 ms and with a pruning ratio of 75% the overhead is 189.8 ms. Thus even if pruning of processes is unbalanced, the amount of pruning has a big impact on the execution time overhead. This can be explained by the difference in the overall amount of debug information that needs to be processed.

E. Case Study

We have evaluated DySectAPI on an MPI bug that only manifested itself at or above 3,456 MPI processes with BoomerAMG, a high-performance preconditioner library developed at the Lawrence Livermore National Laboratory [30].

A specific configuration led to failures at scale and we have used DySectAPI to investigate the issue. An error was emitted by a method within MPI called `MPI_Error_string`. Therefore we started with the simple probe seen in Figure 10 to figure out which call-path led to the error message being printed.

The probe resulted in the debug output shown in Figure 11. From the output we see how a small subset of

```
1  Probe* mpiError =
2    new Probe(Code::location("MPI_Err_str"),
3    Domain::world(100),
4    Acts(Act::stackTrace(),
5         Act::trace("MPI_Err_str probe")));
```

Figure 10: MPI bug probe example.

```
1  DysectAPI Frontend:Info > [6] Trace: MPI
       Error string probe
2  DysectAPI Frontend:Info > [6] Stack trace
3  DysectAPI Frontend:Info > |-> [6] _start >
       __libc_start_main > main >
       HYPRE_PCGSetup > hypre_PCGSetup >
       HYPRE_BoomerAMGSetup >
       hypre_BoomerAMGSetup >
       hypre_seqAMGSetup >
       hypre_BoomerAMGSetup >
       hypre_BoomerAMGCreateS >
       hypre_MatvecCommPkgCreate >
       hypre_MatvecCommPkgCreate_core >
       hypre_MPI_Allgather > MPI_Allgather >
       intra_Allgather > MPIR_ToPointer >
       MPIR_Error > MPIR_Errors_are_fatal >
       PMPI_Error_string
```

Figure 11: MPI bug probe example output.

6 processes called the `MPI_Error_string` routine and had equivalent stack traces. We can also see that the issue could be related to the call to `MPI_Allgather` and its call to `MPIR_ToPointer`, which resolves MPI communicator identifiers into pointers of the corresponding internal communicator structure. To this end, we created a more advanced probe to give more detailed information about the error, as shown in Figure 12. This probe tree captures the callpath from the previous stack trace and prints a trace message with the communicator identifiers. Further, a more detailed segmentation fault detector is setup to show precisely where it happens and to print the communicator identifier.

Figure 13 shows the debugging output of the probe. We see how three processes trigger the segmentation fault exception at the same location. By inspecting the call site at `intra_fns_new.c:2885` we see that this bug only appears in a shared memory feature, sporadically causing floating values to be resolved into pointers. We looked into the communicator initialization code and found a member field that was uninitialized. The problem was an issue in a recently upgraded MVAPICH 1.2.7 MPI library. Based on this diagnosis, a work-around was quickly identified and a bug was reported to the MVAPICH developers.

With this use case, we demonstrate how the prescriptive debugging model can be used as an engine to allow programmers to test their debugging intuition and that the expressiveness of the model helps reduce the error space. The use case shows how the prescriptive debugging model is capable

```
1 DysectAPI Frontend: Info > [369] Trace: MPI_Allgather communicators: comm = [92:92]
2 DysectAPI Frontend: Info > [368] Trace: Intra allgather 1st comm = [92:92]
3 DysectAPI Frontend: Info > [363] Trace: Intra allgather 2nd comm = [92:92]
4 DysectAPI Frontend: Info > [3] Stack trace:
5 DysectAPI Frontend: Info > |-> [3] _start > __libc_start_main > main > HYPRE_PCGSetup >
      hypre_PCGSetup > HYPRE_BoomerAMGSetup > hypre_BoomerAMGSetup > hypre_seqAMGSetup >
      hypre_BoomerAMGSetup > hypre_BoomerAMGCreateS > hypre_MatvecCommPkgCreate >
      hypre_MatvecCommPkgCreate_core > hypre_MPI_Allgather > MPI_Allgather > intra_Allgather
      > intra_Allgather
6 DysectAPI Frontend: Info > [3] Trace: Segmentation fault at location: /usr/local/tools/
      mvapich-intel-debug-1.2/src/src/coll/intra_fns_new.c:2885
7 DysectAPI Frontend: Info > [3] Trace: comm = [0:0]
```

Figure 13: MPI bug advanced probe example output.

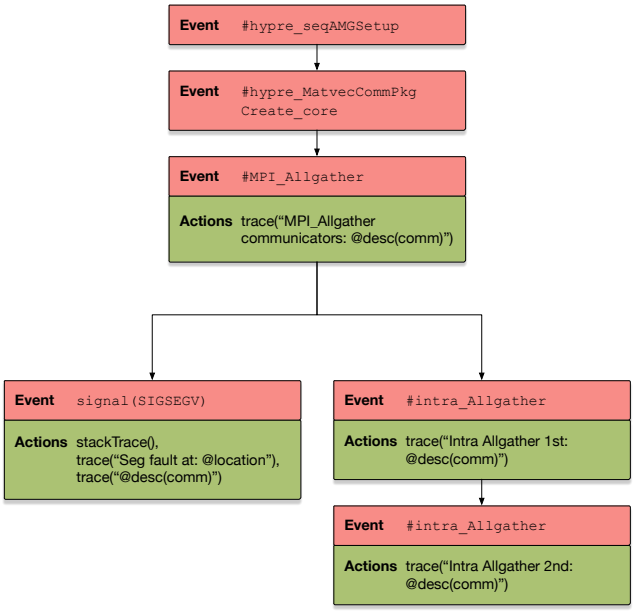


Figure 12: MPI bug advanced probe example.

of reducing to very condensed debugging information even for a complex debugging scenario.

VII. Conclusions

Prescriptive debugging is a novel debugging model that can scale without sacrificing key debugging information presented to programmers, thus filling the gap between traditional and lightweight debuggers. It allows programmers to codify their debugging intuition and to test their hypothesis with minimal user interaction during run-time. This allows the error search space to be reduced such that the information presented to the programmer is very condensed.

Using both experimental results and analytical modeling we show that our prototype implementation, DySectAPI, has logarithmic scaling on current systems. We also predict performance beyond current machine resources, and our model predicts good performance results even for very large system scales.

Acknowledgment

We thank the anonymous reviewers for the valuable comments and suggestions.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-662983). This work has also been partially funded by the ARTEMIS Joint Undertaking as part of the COPCAMS project under GA number 332913.

References

[1] Lawrence Livermore National Laboratory, "Advanced Simulation and Computing Sequoia," https://asc.llnl.gov/computing_resources/sequoia, accessed 17 October 2014.

[2] Oak Ridge National Laboratory, "Introducing Titan - Advancing the Area of Accelerated Computing," https://www.olcf.ornl.gov/titan/, accessed 17 October 2014.

[3] J. Dongarra, P. H. Beckman *et al.*, "The International Exascale Software Project Roadmap," *International Journal of High Performance Computing Applications (IJHPCA)*, 2011.

[4] Rogue Wave Software, "TotalView® Graphical Debugger," http://www.roguewave.com/products/totalview.aspx, 2014, accessed 17 October 2014.

[5] Allinea Software Ltd., "Allinea DDT: The global standard for high-impact debugging," http://www.allinea.com/products/ddt/features, 2014, accessed 17 October 2014.

[6] R. M. Stallman and C. Support, *Debugging with GDB : The GNU source-level debugger*. Free software foundation, 2014.

[7] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[8] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. P. LeGendre, B. P. Miller, M. Schulz, and B. Liblit, "Lessons Learned at 208K: Towards Debugging Millions of Cores," in *Supercomputing (SC)*, 2008.

[9] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable Temporal Order Analysis for Large Scale Debugging," in *Supercomputing (SC)*, 2009.

[10] I. Redbooks, *IBM System Blue Gene Solution: Blue Gene/Q System Administration*, ser. IBM redbooks, 2012.

[11] D. Abramson, I. T. Foster, J. Michalakes, and R. Sosic, "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Communications of the ACM*, 1996.

[12] Q. Gao, F. Qin, and D. K. Panda, "Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *Supercomputing (SC)*, 2007, p. 15.

[13] Rogue Wave Software, "TotalView Achieves Massive Milestone Towards Exascale Debugging," http://www.roguewave.com/company/news-events/press-releases/2012/scalability-milestone-for-totalview-debugger.aspx, accessed 17 October 2014.

[14] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, B. P. Miller, and M. Schulz, "Benchmarking the Stack Trace Analysis Tool for BlueGene/L," in *International Conference on Parallel Computing (ParCo)*, 2007.

[15] Cray, "CrayDoc - ATP 1.7 Man Pages," 2013, accessed 22 December 2014.

[16] M. N. Dinh, D. Abramson, D. Kurniawan, C. Jin, B. Moench, and L. D. Rose, "Assertion Based Parallel Debugging," in *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2011.

[17] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz, "AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks," in *International Conference on Dependable Systems and Networks (DSN)*, 2010.

[18] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Ahn, M. Schulz, and B. Rountree, "Large Scale Debugging of Parallel Tasks with AutomaDeD," in *Supercomputing (SC)*, 2011.

[19] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin, "Accurate Application Progress Analysis for Large-scale Parallel Debugging," in *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[20] S. C. Gupta and G. Sreenivasamurthy, "Navigating "C" in a "Leaky" Boat? Try Purify," www.ibm.com/developerworks/rational/library/06/0822_satish-giridhar/, 2006, accessed 17 October 2014.

[21] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch, "MARMOT: An MPI Analysis and Checking Tool," in *International Conference on Parallel Computing (ParCo)*, 2003.

[22] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: a tool for checking Fortran 90 MPI programs," *Concurrency and Computation: Practice and Experience*, 2003.

[23] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking," in *Supercomputing (SC)*, 2010.

[24] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," in *Supercomputing (SC)*, 2000.

[25] B. Feldman, "Debugging exascale: To heck with the complexity, full speed ahead!" September 2010.

[26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[27] P. Roth, D. Arnold, and B. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *Supercomputing (SC)*, 2003.

[28] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *International Journal of High Performance Computing Applications*, 2000.

[29] D. Arnold, G. Pack, and B. Miller, "Tree-based Overlay Networks for Scalable Applications," *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[30] R. Falgout, T. Kolev, J. Schroder, P. Vassilevski, and U. M. Yang, "Scalable Linear Solvers: Software," https://computation-rnd.llnl.gov/linear_solvers/software.php, 2014, accessed 17 October 2014.